POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

ACA GraphML ZINC project

Author: Paolo Ginefra

Academic year: 2024/25

GitHub Repo: HERE

---

1. Project Goal

The goal of this project is to perform regression on the ZINC dataset. The input data is extracted from the ZINC database and the "Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules" paper, containing about 250,000 molecular graphs with up to 38 heavy atoms. The task is to regress the penalized $logP$ (also called constrained solubility in some works), given by $y = logP - SAS - cycles$, where $logP$ is the water-octanol partition coefficient, SAS is the synthetic accessibility score, and $cycles$ denotes the number of cycles with more than six atoms. Due to computational constraints, the whole project has been developed using a smaller version of the dataset consisting of 10.000 training samples, 1000 validation samples and 1000 test samples.

2. Preface: is it science or engineering?

I am a Computer Science and Engineering student at his first year of Master's (AI track), and I decided to choose this "optional" project as the perfect opportunity to venture in a new niche of ML and to evaluate the new skills I developed in the last year. As my major's name suggests, my background allows me to switch between working as a scientist and working as an engineer, between trying to answer questions and solving problems. I am more research inclined, so when I am learning new topics, I tend to tackle them more in the question-seeking mode. This was my first contact with graphML, but considering the incredible amount of tradeoffs and constraints that I had to overcome, I must consider this more of an engineering project. More in detail, I managed to allocate just **60 man hours** to the development of this project, including the vast "learning the basics" phase necessary to get familiar with graphML. One objective that I decided to pursue with this project was to try for the first time a complete MLOps setup, with a focus on robustness, reproducibility, remote access, and persistent and reliable logging and storing. Even though I had had the chance to use and test all the tools in past experiences, I had never used them all together, and this "integration hell" was probably the biggest setback of this project.
Last but not least **I do not possess a CUDA compatible GPU**, which forced me to use very limiting options like Kaggle that quickly expired. I managed to get a lent computer, but I had access to it for only about **20 hours**.
Given all of these tight non-functional requirements and my novelty to the field, I opted to use a "design thinking" (Fig 1) approach to all the major components and phases of the project. Each time I started from a time-budget constrained research phase, to then extrapolate a few ideas and directions. Those ideas then turned into prototypes that would finally be pruned by testing.
In the end, I converged on some answers to my questions and a solution to the task, but, looking back, I would probably change a lot of my decisions. Luckily, this is simply a sign that this project fulfilled its purpose: I have learned a lot and gained significant experience.
Concluding, please consider this report as the trace of my learning path, more than the description of a final product.

3. Brief introduction to Graph ML

Graph Machine Learning addresses the challenge of applying machine learning techniques to graph-

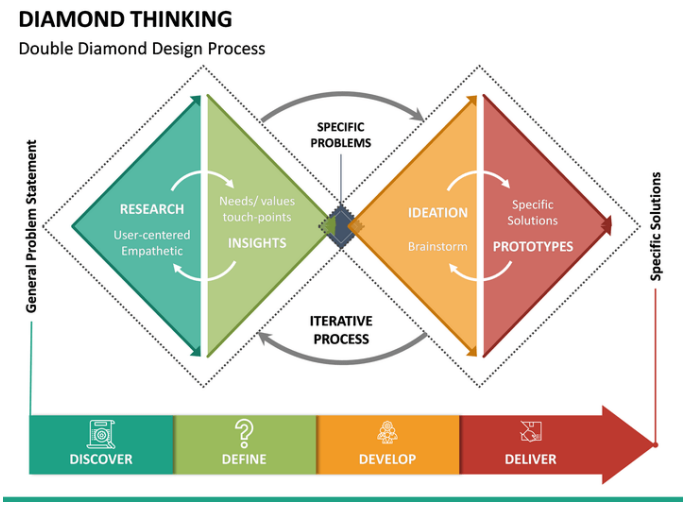

Figure 1: The double diamond "design thinking" method used in this project

structured data, where relationships between entities are as important as the entities themselves. Unlike traditional ML approaches that work with Euclidean data (vectors, images, sequences), graph ML must handle irregular, non-grid structures where each node can have a variable number of neighbors.

The fundamental challenge in graph ML stems from the non-Euclidean nature of graphs. Standard convolutional operations, highly successful in computer vision, cannot be directly applied to graphs due to their irregular topology. This irregularity manifests in several ways: nodes have varying degrees, there is no canonical ordering of neighbors, and the graph structure itself carries meaningful information that must be preserved during learning.

Graph Neural Networks (GNNs) have emerged as the dominant paradigm for learning on graphs. The core principle of GNNs is message passing, where nodes iteratively aggregate information from their neighbors to update their representations. This process allows the network to capture both local neighborhood structures and global graph properties through multiple layers of aggregation.

The field encompasses several key architectures. Graph Convolutional Networks (GCNs) extend the convolution operation to graphs by defining localized filters based on graph structure. GraphSAGE introduces sampling strategies to handle large graphs efficiently. Graph Attention Networks (GATs) incorporate attention mechanisms to weight neighbor contributions dynamically. More recent developments include Graph Transformers and architectures designed for specific graph properties like molecular graphs.

For molecular property prediction, graphs provide a natural representation where atoms become nodes and chemical bonds become edges. This representation preserves the structural information crucial for determining molecular properties, making GNNs particularly well-suited for chemical applications. The challenge lies in designing architectures that can capture the complex relationships between molecular structure and target properties while remaining computationally tractable for large molecular datasets.

4. How to tackle Graph Regression?

The standard phases in solving a general machine learning problem are:

1. Data exploration: get to know the dataset and look for meaningful patterns/issues
2. Data pre-processing: the set of steps to prepare the dataset, which may include normalization, outlier exclusion, data augmentation, feature selection, dimensionality reduction and much more
3. Actual regression: building and training a model to perform regression

Dealing with graph data significantly impacts all the phases, but perhaps the biggest issue has to be extracting structured features from a fundamentally unstructured data type.

As explained in the previous section, the state-of-the-art solution is to use some form of GNNs via Message Passing. A natural research question is then:

How does Message-Passing based feature extraction compare to more traditional methods?

To answer this, I decided to do the regression first using traditional and more "trivial" methods for the feature extraction, to then move to Deep learning ones.

4.1. The traditional stuff

4.1.1 Feature Extraction

In the original dataset, each node can be of one of 28 node types, while each link can represent a single, double, or triple bond. These Node and Edge-level features must be converted to the Graph level. After a research phase and some brainstorming, I narrowed my options to the following feature extraction approaches:

Bag of Nodes (BON) with PCA: This approach treats each molecular graph as a collection of individual node features, creating a histogram-like representation of atom types and their properties. The feature vector is constructed by aggregating node-level attributes across the entire molecule. This naturally produces sparse features; thus, PCA is then applied to reduce dimensionality (and thus memory consumption) while preserving the most significant variance. This representation is particularly useful for capturing the overall atomic composition and electronic properties of molecules, making it effective for predicting properties that depend on the types and quantities of atoms present.

Bag of Edges (BOE): The same concept of BON applied to edges.

Bag of Degrees (BOD): The same concept of BON applied to the node degree (number of connections)
Spectral Features: These features are derived from the eigenvalues of the graph Laplacian matrix, which encode fundamental topological properties of the molecular graph. The Laplacian eigenvalues provide information about graph connectivity, clustering tendencies, and structural symmetries. Spectral features are particularly powerful for capturing global graph properties that are invariant to node ordering, making them valuable for identifying molecules with similar overall topological characteristics regardless of their specific atom arrangements.
Structural Features: This category encompasses traditional graph-theoretic metrics such as clustering coefficient, betweenness centrality, closeness centrality, and graph diameter. These features quantify various aspects of molecular topology, including local clustering patterns, the importance of specific atoms in molecular connectivity, and overall molecular size and shape. Structural features are interpretable and provide direct insights into molecular architecture that correlate with many chemical properties. I chose to extract the number of nodes and edges, the density, and the entropy of the steady state distribution of the molecule.
Master Node Features: This approach augments the original molecular graph by adding a virtual "master node" connected to all atoms in the molecule. Random Walk Positional Embedding is then applied to compute positional encodings for all nodes. The $k-th$ encoding of node n is the probability of getting back to n following a random walk of exactly k steps. The embedding corresponding to the master node is extracted as the feature representation. This method captures global positional information by leveraging the master node's unique perspective of the entire molecular structure. The random walk embedding encodes how information flows through the molecular graph, providing a global representation that considers both local neighborhoods and long-range connectivity patterns.

4.1.2 Regressors

To evaluate the effectiveness of each feature extraction method, I chose to do an extrinsic evaluation by using simple regressors to tackle the ZINC dataset. I chose to use the validation mae to then perform feature selection. I selected four diverse regression algorithms that represent different learning paradigms and assumptions about the underlying data relationships:
LASSO (Least Absolute Shrinkage and Selection Operator): This linear regression method incorporates L1 regularization, which automatically performs feature selection by driving less important coefficients to zero. LASSO is particularly valuable in this context because it provides interpretable linear relationships between molecular features and the target property (penalized logP).
Random Forest: This ensemble method combines multiple decision trees through bootstrap aggregation, providing robust predictions that are less sensitive to outliers and noise. Random Forest naturally handles feature interactions and non-linear relationships without requiring explicit feature engineering.
XGBoost (Extreme Gradient Boosting): This gradient boosting framework builds an ensemble of weak learners sequentially, with each new model correcting the errors of previous ones. XGBoost is known for its superior performance on structured data and its ability to capture complex nonlinear patterns. Its built-in regularization mechanisms help prevent overfitting while maintaining high predictive accuracy.
Support Vector Regression (SVR): This method extends support vector machines to regression tasks by finding a hyperplane that best fits the data within a specified margin of tolerance. SVR is effective at handling high-dimensional feature spaces and can model both linear and non-linear relationships through kernel functions.
The combination of these four algorithms provides comprehensive coverage of different modeling approaches: linear methods (LASSO), tree-based ensembles (Random Forest, XGBoost), and kernel methods (SVR). This diversity ensures that the feature evaluation process captures various types of relationships between molecular structure and target properties, leading to more robust feature selection decisions.

4.1.3 Feature Selection

The world of feature selection methods is vast. I focused on simplicity by looking at **Wrapper methods** and in particular at **Forward Feature Selection**. Given the current selected features, all the remaining ones are added, tested, and removed, keeping only the one that increases the performance the most. This process repeats until there are no more features left or the model stops improving.

4.2. Deep Learning Pipeline

Once again, the choice space is vast and intricate. Being new to the field, I got a bit overwhelmed by all the options for choosing an architecture, especially by all the recent GNN layer types. Having not much domain knowledge (both chemical and in GML), I decided to define a meta architecture, meticulously parametrize it to handle many different layer types and configurations, and run an extensive Hyperparameter Optimization run. I will now list all the pieces and options I decided to test.

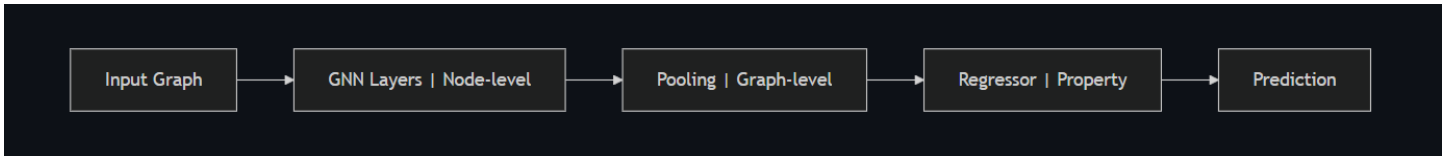

Figure 2: The meta level definition of the pipeline

4.2.1 Meta Architecture

As a Meta Architecture, I decided to develop a Pipeline inspired by Convolutional Neural Networks. CNN works by concatenating a series of convolutional and pooling layers to extract relevant and structured features, to then adding regression layers on top. GNNs can function similarly. You can use Message Passing layers to extract node/edge level features, Pooling layers to pool together the graph level features, and finally use standard regression layers. Even though hierarchical architectures are on the rise (where you alternate Message Passing and Pooling to gradually arrive at the graph level features), I decided to keep the investigation on stacking for a later project by focusing on just one pooling operation. You can find a high-level view at Fig 2. I will now discuss all the different options that I chose for each component.

4.2.2 Message Passing (GNN)

I implemented a flexible GNN architecture supporting 13 different layer types, organized by computational complexity. Table 1 summarizes each layer's mechanism and characteristics.

4.2.3 Pooling

For the Pooling I have chosen 4 types of pooling ranging in complexity you can find them in table 2.

4.2.4 Regressor

For the Regressor, I have chosen a trivial one (Linear layer) and several variations of the Multi-Level Perceptron: one including residual connections and an ensemble.

4.2.5 Data Normalization and Augmentation

Before running the networks, I wanted to make sure that the incoming data was at its maximum potential. To do so, I chose to do:

1. **Normalization**: z-score on the training targets (remove mean and scale with std)
2. **Outlier removal**: remove datapoints whose target is more than 3 stds from the mean
3. **Data augmentation**: add the Master Node, and then for each node include the steady state distribution, the Laplacian eigenvalues, and the Random walk positional encodings
4. **Dimensionality Reduction**: use PCA to project the feature in a smaller, more meaningful space

4.2.6 Hyperparameter optimization

Hyperparameter optimization is the process of systematically searching for the best set of configuration variables, called hyperparameters, that control the learning behavior and architecture of machine learning models. These hyperparameters are chosen before training, and their optimal values are not learned from data but must be determined through experimentation. Methods like those usually belong to the field of gradient-free optimization, and some examples are Bayesian optimization, evolutionary algorithms, or tree-structured Parzen estimators (TPE). I chose the latter.
TPE models the probability distribution of hyperparameters given the performance of previous evaluations. Specifically, it divides the search history into two groups: those with "good" performance and those with "bad" performance, based on a quantile threshold. It then models the hyperparameter distributions for these groups using kernel density estimation (Parzen windows), allowing the algorithm to estimate the likelihood that a new set of hyperparameters will yield improved performance.
I decided to run a **MULTI OBJECTIVE Hyperparameter optimization** by minimizing:

- **Validation MAE**
- **Number of trainable parameters**
- **Training time (time/epoch)**
- **Inference Latency (time/batch)**

5. Tools and implementation details

The whole project is implemented using PyTorch, in particular adding PyTorch Geometric for the graph data handling and the various GNN layers implementation. Unfortunately, this was only the beginning of my MLOps journey because if training takes time, hyperparameter tuning takes ages. I needed to develop a framework to ensure the training goes reliably and robustly. Because of the very tight time constraints, expensive failures could generate a lot of damage to the project development. Also, not having easy access

to a GPU locally, I needed to produce easily deployable code. In this section, I will list some of the tools I used and the choices I made to achieve my goals (I will focus more on the machine learning related tools).

Pytorch Lightning - Reliable PyTorch integration PyTorch Lightning is a high-level framework that organizes and simplifies deep learning workflows built with PyTorch, making it easier to write clean, reproducible, and scalable code. It abstracts away boilerplate such as training loops, logging, and device management, while maintaining full flexibility to use raw PyTorch where needed. This allows users to focus on model design and experimentation, enables effortless multi-GPU or distributed training, and provides built-in features for logging, checkpointing, and early stopping. As a result, PyTorch Lightning is widely used for accelerating research, prototyping, and production deployment of deep learning models

Weights and Biases - Persistently track training Weights & Biases (W&B) is a comprehensive MLOps platform designed to help data scientists and machine learning engineers track, visualize, and manage machine learning experiments and workflows. It enables automatic logging of training metrics, hyperparameters, and model artifacts, providing a centralized, interactive dashboard for real-time monitoring and comparison of different runs. W&B is useful for experiment reproducibility, efficient hyperparameter optimization, and model versioning, making it easier to iterate on models, share results with teams, and streamline the path from experimentation to deployment.

Optuna - hyperparameter optimization Optuna is an open-source, automatic hyperparameter optimization framework designed for machine learning. It allows users to efficiently search for the best set of model hyperparameters by leveraging state-of-the-art algorithms such as TPE, random search, and Bayesian optimization. Optuna can integrate with a remote MySQL database to persistently log the results. I hosted mine for free on SupaBase. Via the package optuna-dashboard, it is also possible to connect to the db remotely, visualizing a real-time dashboard of the logged studies.

Training Hardware The laptop that I eventually used has a **NVIDIA GeForce RTX 3050** GPU with just **4GB** of dedicated VRAM.

Implementation philosophy I designed the bulk of my code as a Python package in order to be easily installable via pip on another machine and/or in a virtual environment. All the analyses are then performed using Python Notebooks.
For more details regarding the implementation, please refer to the GitHub repository.

6. Analysis and Results

In this section, I will present the analysis and the results I obtained. I will report only the major choices and findings as well as some improvements that can be made to the process. For more details, refer to the plots in the Appendix (Sec 8) or the notebooks in the GH repository.

6.1. Data Exploration

The major findings of the data exploration are :

- **Target imbalances**: The target distribution is not ideal for learning. It is highly concentrated around the mode. This would suggest the usage of some resampling techniques, but this was skipped due to time constraints.
- **Target outliers**: The targets have a few strong outliers whose targets are up to 20σ away from the mean. They must be removed before training to massively simplify the problem. Also, plotting the outliers, I couldn't find any specific pattern relating them. Some more in-depth feature comparison might reveal some more information.
- **Node and Edge feature imbalances**: both the Node and Edge level features present some strong imbalances, but this has to be expected and should not be an issue for training.
- **Bimodal node count and edge count distributions**: Both the node count and edge count distributions present a clear bimodal nature. This could mean that the dataset is mixing two processes. Some Gaussian Mixture Clustering analysis in the feature space might reveal some info

6.2. Traditional Feature-Based Regression

The number and groups of extracted features can be seen in Table 3.

6.2.1 Which feature group is more useful for regression?

The performance of each group with each regressor can be seen in Fig 11. The **Bag of Node** features clearly dominate all the others, achieving the best **MAE of 0.366**. This suggests that just the distribution of the node types gives significant information regarding the $logP$ of a given molecule.

6.2.2 What can we infer from the performances of the various regressors?

Interestingly **SVR** performs consistently similarly to **XGB** suggests that the more a pattern is present the more close graph in feature space tend to have close $logP$

6.2.3 What are the results of feature selection and what can they tell us?

The selected features can be seen in Table 4 and a comprehensive plot summary can be found in Fig 13. The first two features to be added are a **BON** and a **BOE**, suggesting that a big distinction can be made just from the presence of a certain combination of nodes and the number of double bonds. Also the only Master Node feature included are the 7 and 8 hops encoding. I suspect this is linked to a correlation between the random walk encodings and the number of cycles with that "hops" number of nodes. This makes even more sense looking at the formula of the $logP$.

6.2.4 What can the visualization of the feature space tell us?

In Fig 15 you can find a tSNE embedding of the feature space. This kind of embedding scheme aims to preserve the local similarity between data points. The smooth target gradient reinforces the points made from the SVR.
Surprisingly, there seem to be some outliers to the smoothness, which are consistently lower than the average of their neighbours. This is also abundantly clear, looking at the prediction error coloring.
This is also confirmed by the second band in Fig 14.
There can be many explanations for these outliers, ranging from simply being there to a higher-level pattern that is not captured by the features to being some mistakes.
In any case, this phenomenon in the dataset (a complex pattern that most of the time behaves like a simple one) makes learning the complex pattern extremely difficult. This challenge is way over scope for this project, but can be further investigated.

6.3. Deep Learning

The number of choices and details regarding this section is astronomical. I will just report the ones most significant to me. Also, a complete and in-depth statistical analysis has been performed on the results, the produced visualizations are in the APPENDIX (Section 8.3)

6.3.1 Goals of Hyperparameter optimization

The objectives to be minimized are:

1. Best validation Mean Absolute Error (MAE)
2. Total Number of trainable parameters
3. GNN Total Number of trainable parameters
4. Pooling Total Number of trainable parameters
5. Regressor Total Number of trainable parameters
6. Training time [min/epoch]
7. Inference latency [ms/batch] (All the trainings are done with data batches of 32 elements)

6.3.2 Hyperparameter optimization budget

Each trial of the hyperparameter optimization has been trained for a maximum of 45 epochs. And the whole run for about 12 hours. These resulted in **123 trials**. This very limited training budget will inevitably introduce some strong bias towards simpler architectures and higher learning rates

6.3.3 Hyperparameter optimization hyperparameters

All the hyperparameters considered in the optimization can be found in Table 5.

6.3.4 What is the best model configuration?

The top 3 best configurations can be found in Table 6

6.3.5 Between GNN, pooling, and regression, which is more important?

From the statistical analysis, it appears very clearly that the **GNN dominates** everything else, both in terms of the number of parameters and correlation with performance. Also, it appears that attention-based GNNs tend to perform slightly better. For the pooling, changing the type doesn't influence the MAE much; thus, the preferred type should be 'mean', limiting model parameters. The same story goes for the regressor; just pick a linear one.
These findings confirm once again that this is primarily a task of high-quality feature extraction rather than pure regression.

6.3.6 What can we say about the embeddings produced by the model?

Looking at Fig 16, especially comparing to the traditional features in Fig 15, it seems clear that the best identified model is fully capable of extracting meaningful and locally smooth embeddings of the graphs, capturing the overall pattern meaningfully. The problems with the outliers previously discussed still hold, and this is definitely further proof that a more complex pipeline is present overshadowed by a majority of simpletons

7. Conclusion

In this project, I implemented a complete and robust MLOps system aimed at solving the ZINC Dataset Regression. In the meantime, I extracted useful insights in the world of GNNs. In my opinion, much more can be done with this project. The biggest mystery remains the outlier problem, and to solve it, I think the best way would be to run all the analysis on the full Dataset and not on the reduced one. This would require more computational time and more powerful hardware.

By my analysis so far, it seems that the problem with this small dataset can be considered solved. One can try to resample the dataset to handle the target imbalance or try and run more complex models for more time, but I believe that as long as a simple explanation exists for the majority of the dataset, the local minima in the loss landscape will be too enticing for every model, especially if regularization techniques are in place.

Layer Type	Description	Formula
GCN	Does not support edge features. Degree-normalized neighborhood aggregation. Simple and interpretable, suitable for basic graph tasks.	$\mathbf{H}^{(l+1)} = \sigma(\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}\mathbf{H}^{(l)}\mathbf{W}^{(l)})$
SGConv	Does not support edge features. Simplified GCN by removing nonlinear activations. Fast inference for shallow architectures.	$\mathbf{H}^{(l+1)} = (\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2})^K\mathbf{H}^{(l)}\mathbf{W}$
GraphConv	Does not support edge features. Flexible aggregation and serves as a versatile starting point.	$\mathbf{h}_u^{(l+1)} = f(\mathbf{W}_1\mathbf{h}_u^{(l)} + \sum_{v\in\mathcal{N}(u)}\mathbf{W}_2\mathbf{h}_v^{(l)})$
SAGE	Does not support edge features. Samples fixed-size neighbors for scalability, efficient for large graphs.	$\mathbf{h}_u^{(l+1)} = f(\mathbf{W}_1\mathbf{h}_u^{(l)} \Vert \mathbf{W}_2 \cdot \mathrm{AGG}(\{\mathbf{h}_v^{(l)}\}))$
GINConv	Does not support edge features. Maximally expressive for graph isomorphism. Ideal for structural distinction.	$\mathbf{h}_u^{(l+1)} = \mathrm{MLP}((1+\epsilon)\mathbf{h}_u^{(l)} + \sum_{v\in\mathcal{N}(u)}\mathbf{h}_v^{(l)})$
ChebConv	Does not support edge features. Spectral convolution with Chebyshev polynomials. Captures localized patterns.	$\mathbf{H}^{(l+1)} = \sum_{k=0}^{K}\mathbf{T}_k(\tilde{\mathbf{L}})\mathbf{H}^{(l)}\mathbf{W}_k$
ARMAConv	Does not support edge features. Smooth surface modeling via autoregressive filters.	$\mathbf{H}^{(l+1)} = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\mathbf{V}_k^{(l)}\right)$
TAGConv	Does not support edge features. Adapts to local topology diversity with K-hop filters.	$\mathbf{H}^{(l+1)} = \sigma\left(\sum_{k=0}^{K}\mathbf{A}^k\mathbf{H}^{(l)}\mathbf{W}_k^{(l)}\right)$
GAT	Supports edge features. Attention-based aggregation to learn dynamic neighbor importance.	$\mathbf{h}_u^{(l+1)} = \sigma\left(\sum_{v\in\mathcal{N}(u)}\alpha_{uv}\mathbf{W}\mathbf{h}_v^{(l)}\right)$ $\alpha_{uv} = \mathrm{softmax}(\mathrm{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{h}_u \Vert \mathbf{W}\mathbf{h}_v]))$
GATv2	Supports edge features. Improved GAT with nonlinear projection before scoring.	$\alpha_{uv} = \mathrm{softmax}(\mathbf{a}^T\mathrm{LeakyReLU}(\mathbf{W}[\mathbf{h}_u \Vert \mathbf{h}_v]))$
Transformer Conv	Supports edge features. Self-attention mechanism for long-range dependencies.	$\mathbf{h}_u^{(l+1)} = \mathbf{W}_1\mathbf{h}_u^{(l)} + \sum_{v\in\mathcal{N}(u)}\alpha_{uv}\mathbf{W}_2\mathbf{h}_v^{(l)}$ $\alpha_{uv} \propto \exp(\mathbf{q}_u^T\mathbf{k}_v/\sqrt{d})$
GINEConv	Supports edge features. Extends GIN with edge features for molecular graphs.	$\mathbf{h}_u^{(l+1)} = \mathrm{MLP}((1+\epsilon)\mathbf{h}_u^{(l)} + \sum_{v\in\mathcal{N}(u)} f(\mathbf{h}_v^{(l)}, \mathbf{e}_{uv}))$
PNA	Supports edge features. Combines multiple aggregators with degree scaling. Highly expressive.	Not relevant

Table 1: GNN Layer Types with Descriptions and Formulas. Complexity is indicated by row color: green for low, orange for medium, blue for medium-high, and red for high.

Pooling Type	Description	Output Dimension	Formula
Mean	Computes the element-wise mean (average) of all node feature vectors in a given set or graph.	hidden_dim	$z = \frac{1}{\|V\|} \sum_{v \in V} h_v$
Max	Computes the element-wise maximum value across all node feature vectors in a set or graph.	hidden_dim	$z = \max_{v \in V} h_v$ (per dimension)
Attentional	Applies a learned attention mechanism to weight each node's contribution before aggregation.	hidden_dim	$z = \sum_{v \in V} a_v h_v$, $a_v = \text{softmax}(f(h_v, q))$ q: query vector
Set2Set	Uses a recurrent neural network to process node features as a set, iteratively building a context vector.	2× hidden_dim	Iterative process: $q_t = \text{LSTM}(o_{t-1}, q_{t-1})$, $a_{v,t} = \frac{\exp(f(h_v, q_t))}{\sum_{u \in V} \exp(f(h_u, q_t))}$, $o_t = \sum_{v \in V} a_{v,t} h_v$, Final output: $[q_T, o_T]$

Table 2: Pooling Types in Graph Neural Networks with Descriptions and Formulas. Row colors highlight key properties: green for basic operations, orange for moderately complex methods, and blue for advanced techniques.

Feature Group	Number of Features	Features
BON	10	`BON_PC1, BON_PC2, BON_PC3, BON_PC4, BON_PC5, BON_PC6, BON_PC7, BON_PC8, BON_PC9, BON_PC10`
BOE	3	`BOE_1, BOE_2, BOE_3`
BOD	4	`BOD_deg_0, BOD_deg_1, BOD_deg_2, BOD_deg_3`
Spectral	8	`SPECTRAL_1, SPECTRAL_2, SPECTRAL_3, SPECTRAL_4, SPECTRAL_5, SPECTRAL_6, SPECTRAL_7, SPECTRAL_8`
Structural	4	`STRUCT_num_nodes, STRUCT_num_edges, STRUCT_density, STRUCT_ss_entropy`
MasterNode	11	`MASTER_RWPE_1, MASTER_RWPE_2, MASTER_RWPE_3, MASTER_RWPE_4, MASTER_RWPE_5, MASTER_RWPE_6, MASTER_RWPE_7, MASTER_RWPE_8, MASTER_RWPE_9, MASTER_RWPE_10, MASTER_RWPE_11`

Table 3: Feature groups and their corresponding features.

Step	Feature Added	MAE	Improvement
1	BON_PC8	0.6114	0.1956
2	BOE_1	0.5200	0.0914
3	BON_PC5	0.4548	0.0653
4	BON_PC9	0.4132	0.0416
5	BON_PC10	0.3796	0.0336
6	BON_PC1	0.3654	0.0142
7	BON_PC4	0.3460	0.0194
8	MASTER_RWPE_8	0.3350	0.0110
9	BOD_deg_3	0.3282	0.0068
10	MASTER_RWPE_7	0.3245	0.0037
11	BOD_deg_2	0.3196	0.0049
12	No improvement found. Stopping.		

Table 4: Feature selection steps with corresponding MAE and improvement values.

Table 5: Hyperparameter Optimization Parameters for GNN Pipeline

Parameter	Type	Possible Values	Purpose
GNN Architecture			
`hidden_dim`	Categorical	{64, 128, 256}	Hidden dimension size for GNN layers, controls model capacity and representation power
`num_layers`	Integer	[3, 6]	Number of GNN layers, affects model depth and ability to capture long-range dependencies
`layer_name`	Categorical	{'GINEConv', 'GAT', 'GATv2', 'SAGE', 'PNA', 'ChebConv', 'GCN', 'TransformerConv'}	Type of GNN layer, determines message passing mechanism and inductive bias
Regularization			
`dropout_rate`	Float	[0.1, 0.3] (step=0.05)	Dropout rate for GNN layers to prevent overfitting and improve generalization
`global_dropout`	Float	[0.0, 0.2] (step=0.05)	Global dropout applied throughout the model for additional regularization
Learning Parameters			
`lr`	Float (log)	[1e-4, 5e-3]	Learning rate for optimizer, controls convergence speed and stability
`weight_decay`	Float (log)	[1e-6, 1e-3]	L2 regularization strength to prevent overfitting
Pooling Strategy			
`pooling_type`	Categorical	{'mean', 'attentional', 'set2set'}	Graph-level pooling method for aggregating node representations to graph representation
Regressor Architecture			
`regressor_type`	Categorical	{'linear', 'mlp', 'residual_mlp', 'ensemble_mlp'}	Type of regression head for final prediction
`regressor_num_layers`	Integer	[1, 3]	Number of hidden layers in MLP regressor (conditional on regressor type)
`regressor_first_hidden`	Categorical	{128, 256, 512}	Hidden dimension of first regressor layer (conditional on MLP types)
`regressor_second_hidden`	Categorical	{64, 128, 256}	Hidden dimension of second regressor layer (conditional on $\geq$2 layers)
`regressor_third_hidden`	Categorical	{32, 64, 128}	Hidden dimension of third regressor layer (conditional on 3 layers)
`regressor_dropout`	Float	[0.0, 0.3] (step=0.05)	Dropout rate specific to regressor layers
`regressor_normalization`	Categorical	{'batch', 'none'}	Normalization method applied in regressor layers
Fixed Parameters (Not Optimized)			
`optimizer`	Fixed	'adamw'	Optimizer type for training
`lr_scheduler`	Fixed	'cosine'	Learning rate scheduling strategy
`loss`	Fixed	'mae'	Loss function (Mean Absolute Error)
`warmup_epochs`	Fixed	5	Number of warmup epochs for learning rate scheduling
`gradient_clip_val`	Fixed	1.0	Gradient clipping value to prevent exploding gradients

Table 6: Top 3 Trials Based on Validation MAE with Hyperparameters and Objectives

Metric / Hyperparameter	**Trial #106 (Rank 1)**	**Trial #80 (Rank 2)**	**Trial #117 (Rank 3)**
Validation MAE	0.1787	0.1861	0.1887
Total Parameters	812,289	430,530	1,010,049
GNN Parameters	812,160	426,240	812,160
Pooling Parameters	0	4,225	197,632
Regressor Parameters	129	65	257
Training Time (min/epoch)	0.1806	0.1354	0.2181
Inference Latency (ms)	15.36	10.67	12.44
GNN Architecture			
Layer Type	GATv2	TransformerConv	GATv2
Hidden Dimension	128	64	128
Number of Layers	3	4	3
Dropout Rate	0.1	0.1	0.1
Learning			
Learning Rate	1.31e-3	2.11e-3	7.66e-4
Weight Decay	4.65e-6	2.46e-6	1.62e-6
Global Dropout	0.0	0.0	0.15
Pooling			
Pooling Type	mean	attentional	set2set
Regressor			
Regressor Type	linear	linear	linear

8. APPENDIX - Plots

In this Appendix I will mostly put the plots I produced in this project

8.1. Data Exploration

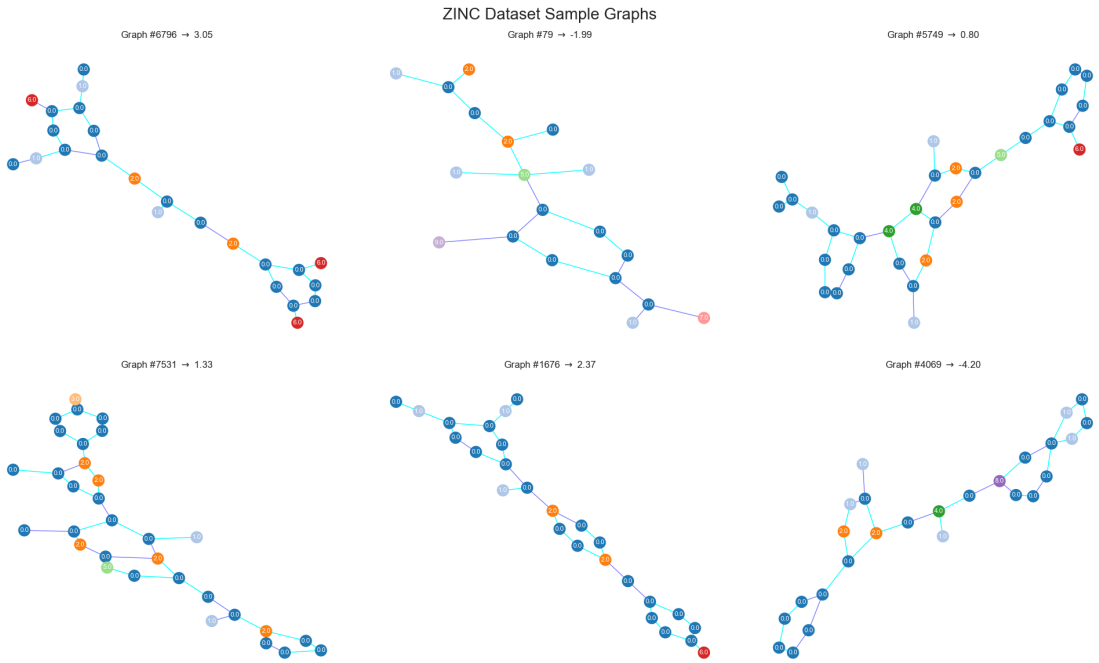

Figure 3: Plotting some of the molecules of the dataset

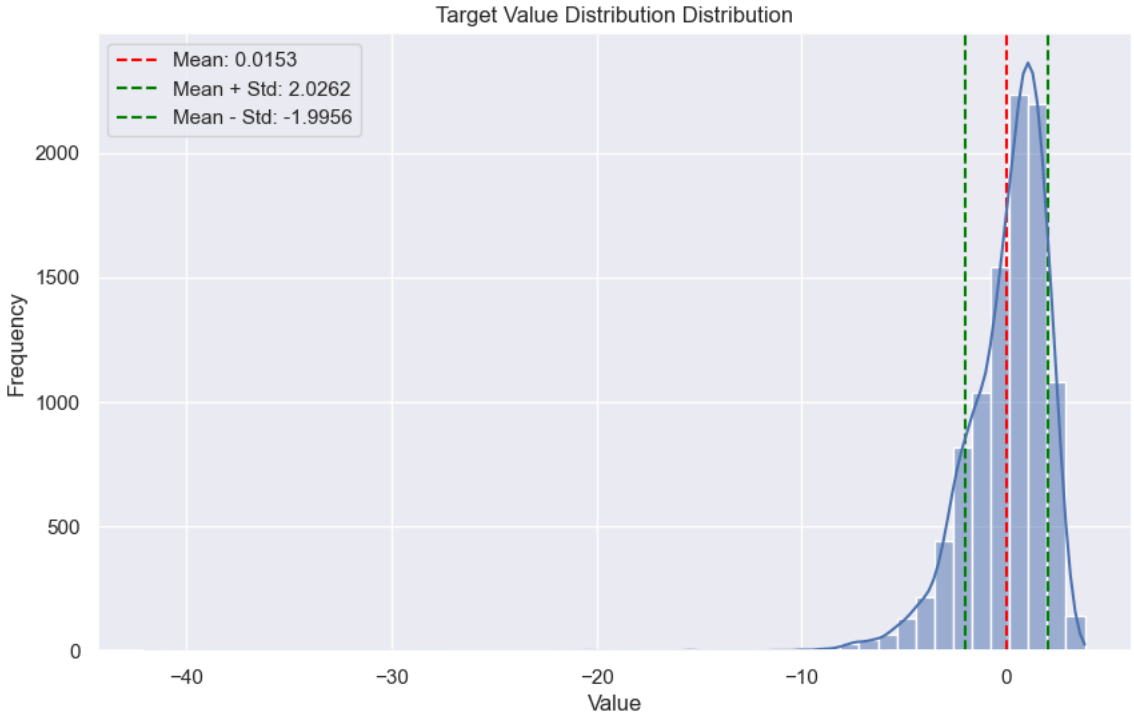

Figure 4: The distribution of the training dataset targets

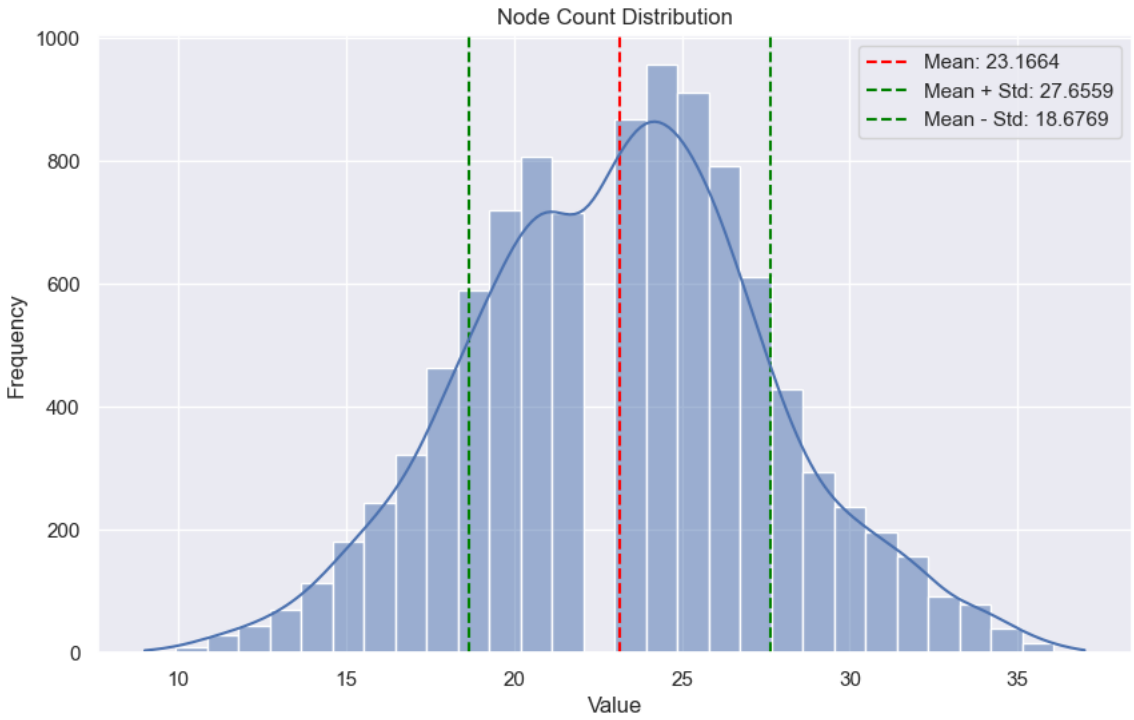

Figure 5: The muber of nodes per graph distribution of nodes across the dataset

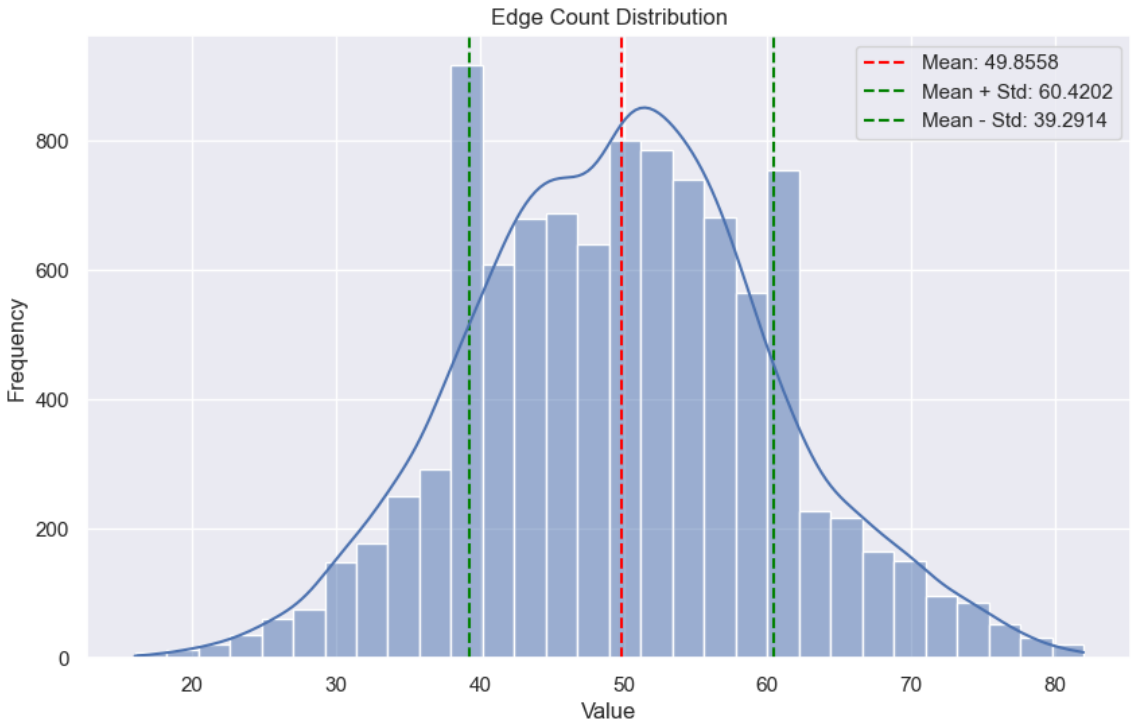

Figure 6: The distribution of edge counts in the dataset

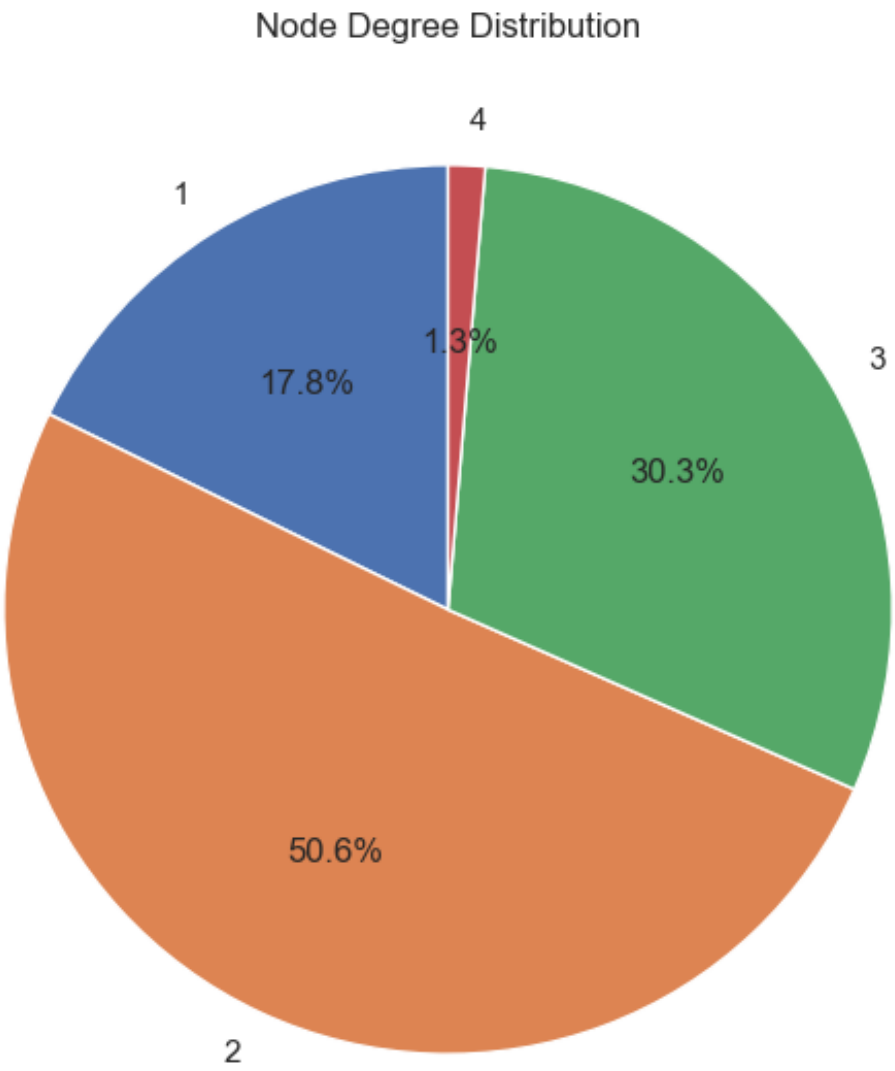

Figure 7: The distribution of node degrees in the dataset

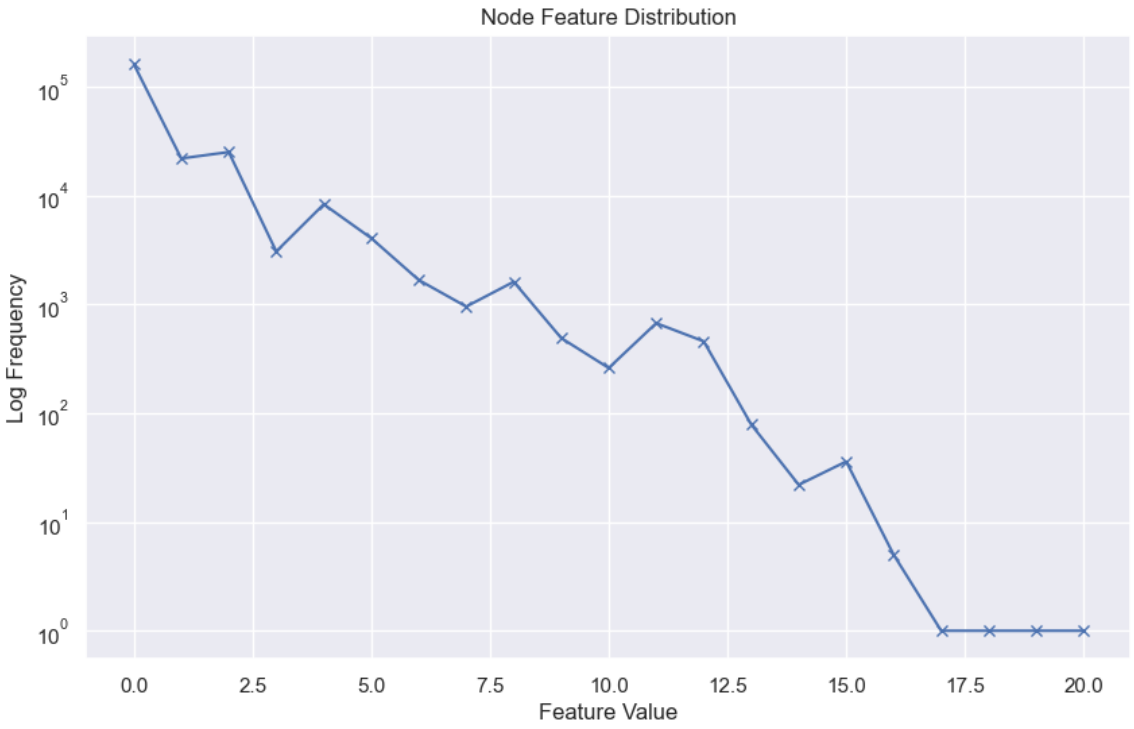

Figure 8: The distribution of node features in the dataset

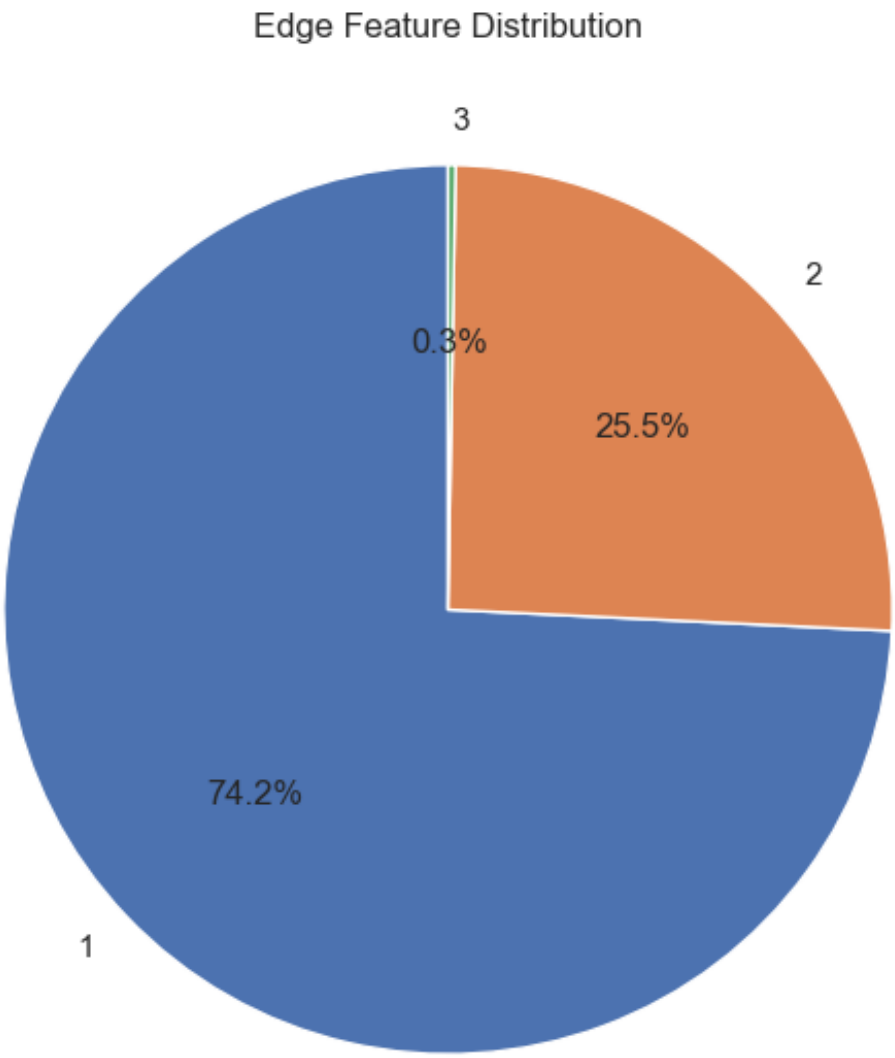

Figure 9: The distribution of edge features in the dataset

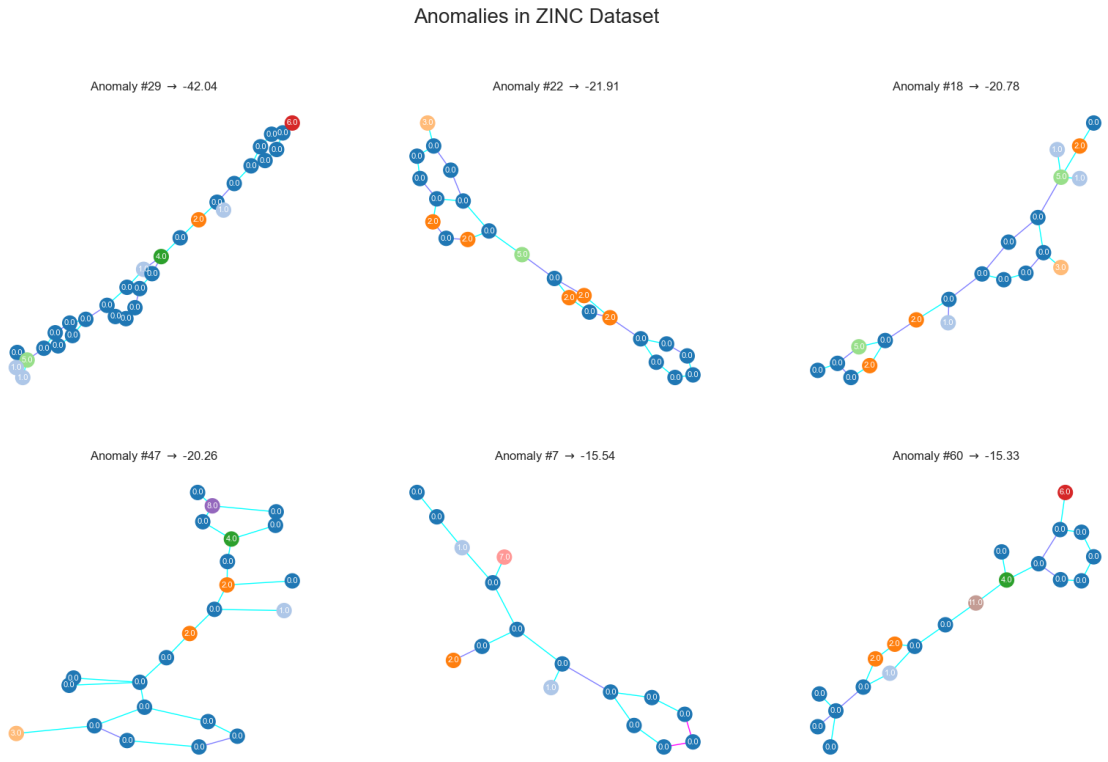

Figure 10: Plots highlighting the top anomalies in the dataset (more than 3 stds away from the mean ranked by distance from the mean)

8.2. Traditional Feature-Based Regression

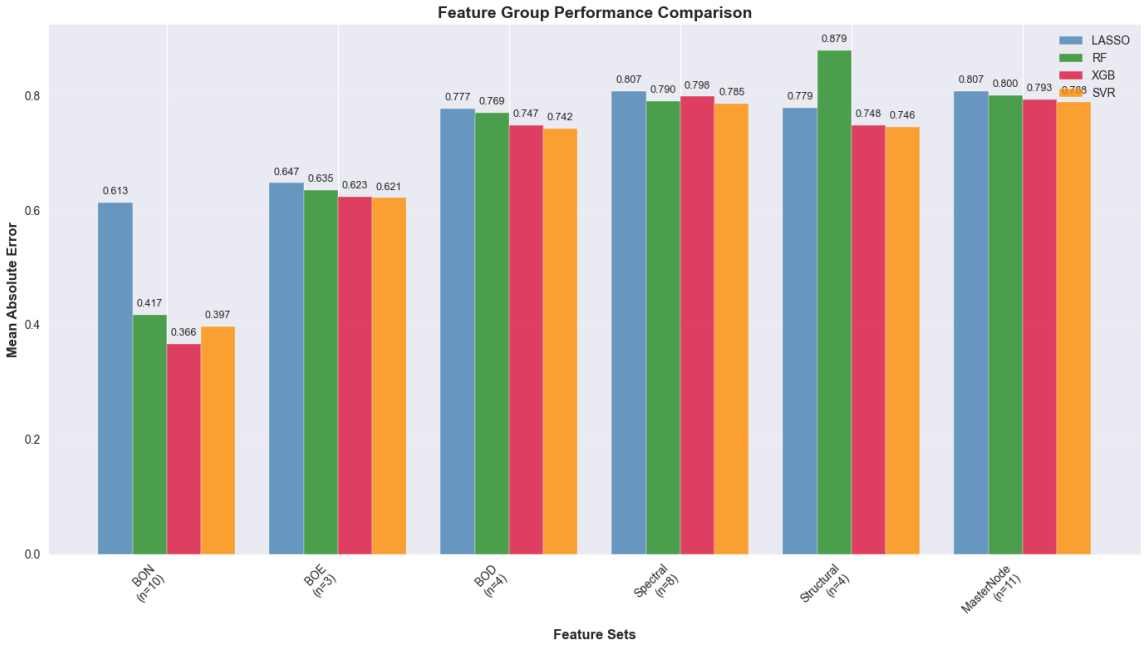

Figure 11: Feature Groups Validation MAE with the various regressors

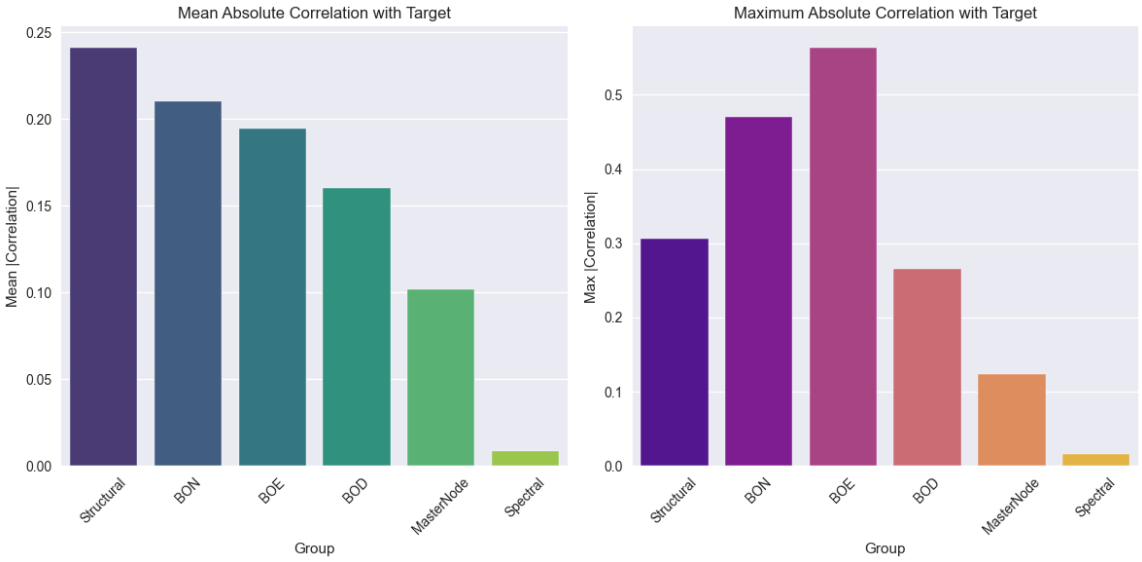

Figure 12: Target feature Correlations per group

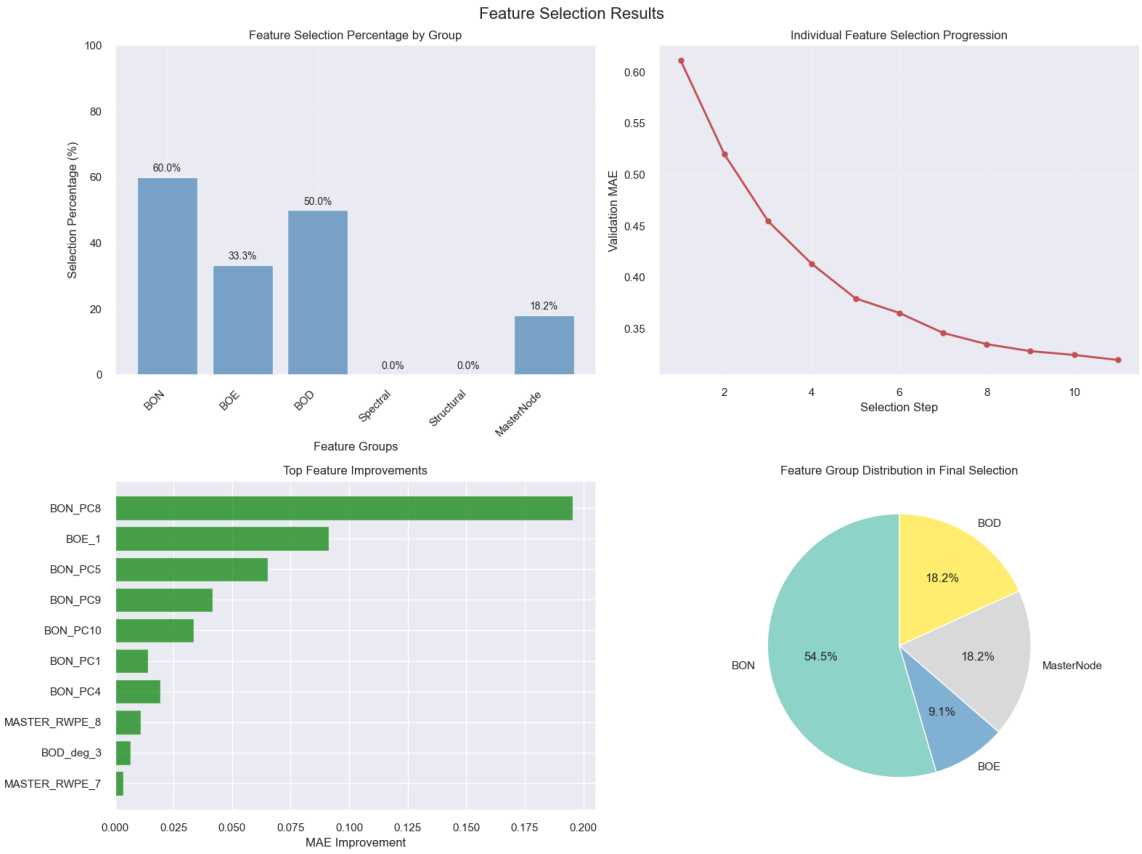

Figure 13: Results of the feature selection process

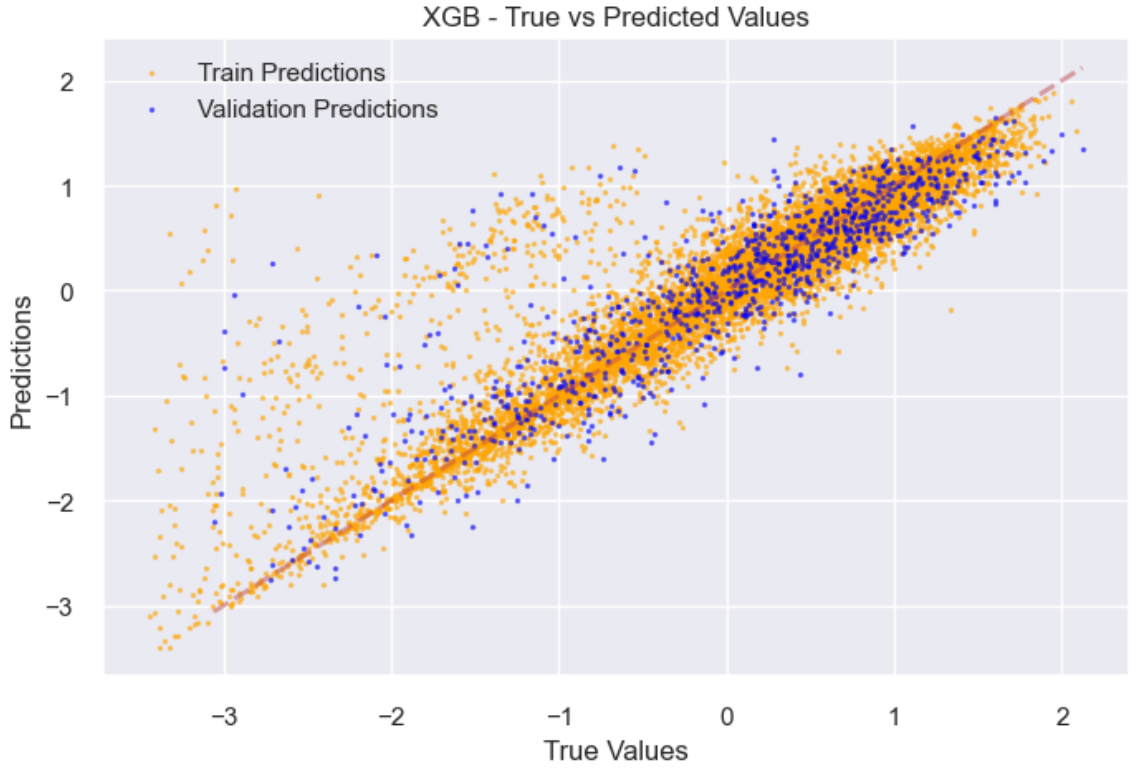

Figure 14: Comparison of true vs. predicted values in the final best model on the selected features

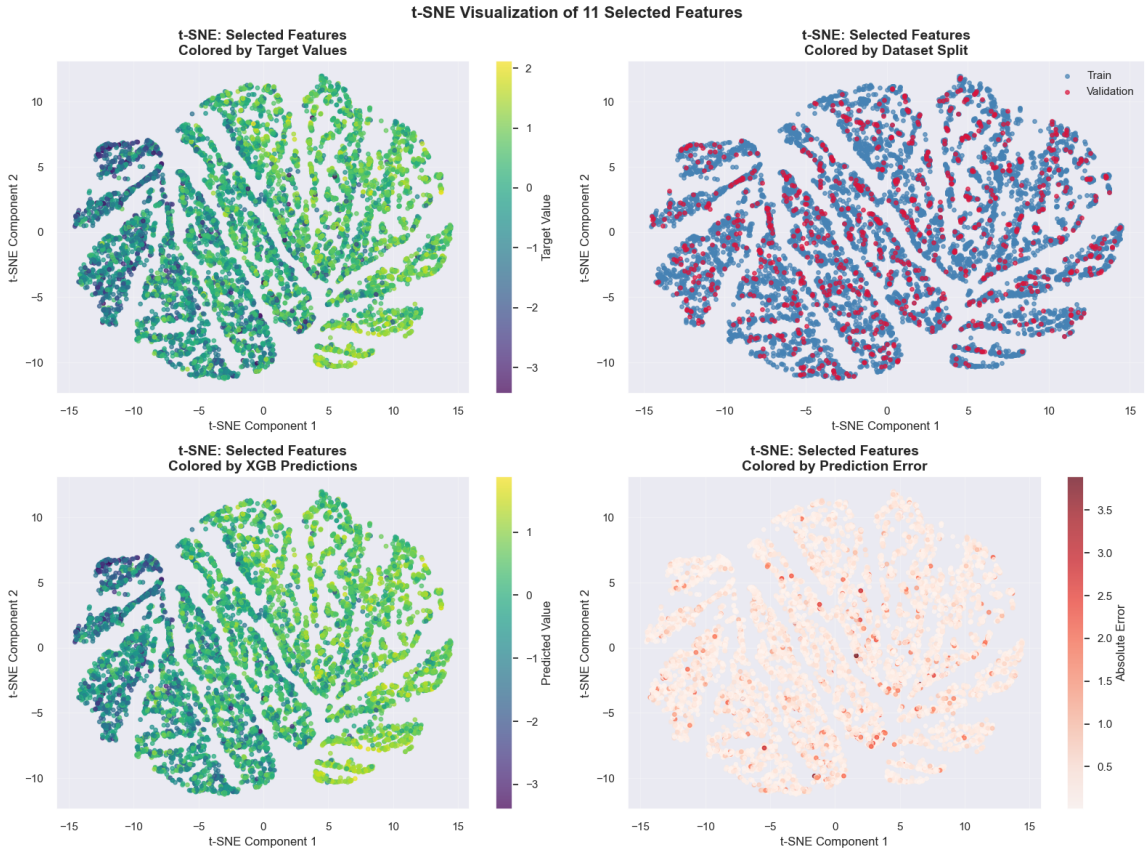

Figure 15: Tsne of the features colored by different metrics

8.3. Deep learning

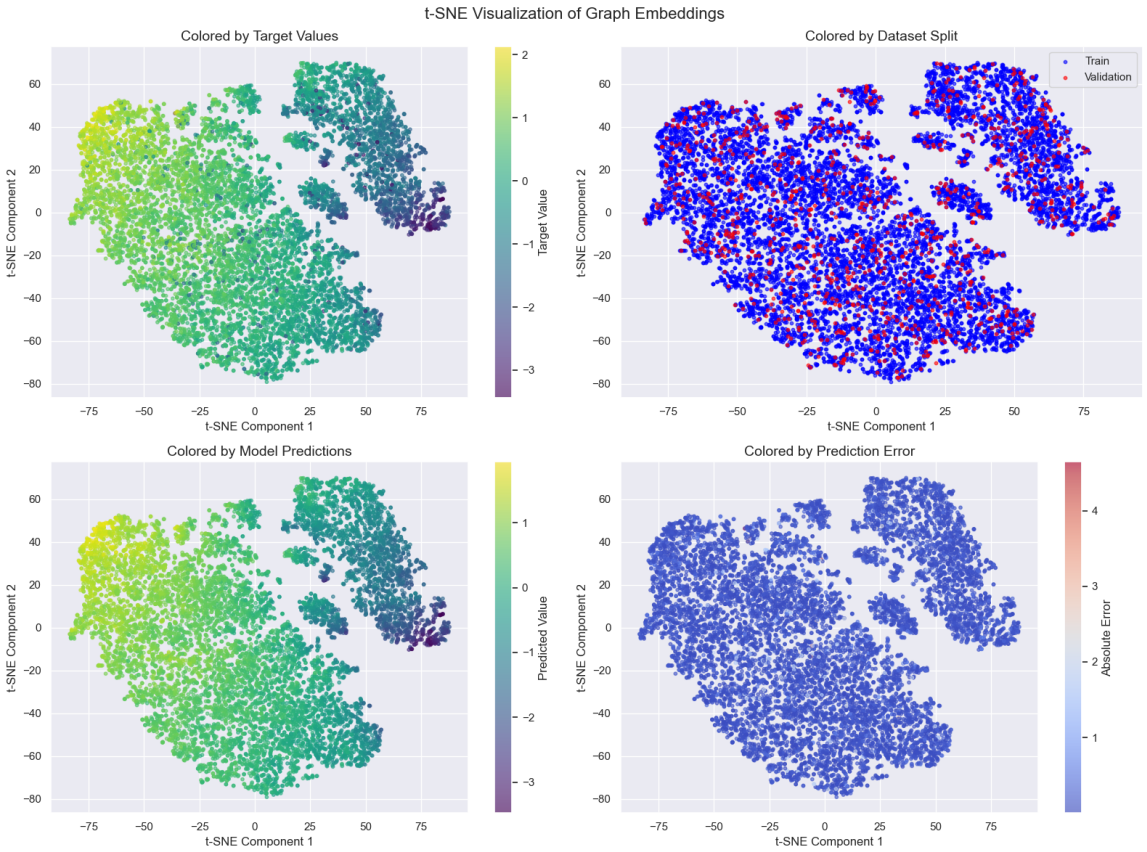

Figure 16: The Feature extracted by the best model reduced with tSNE

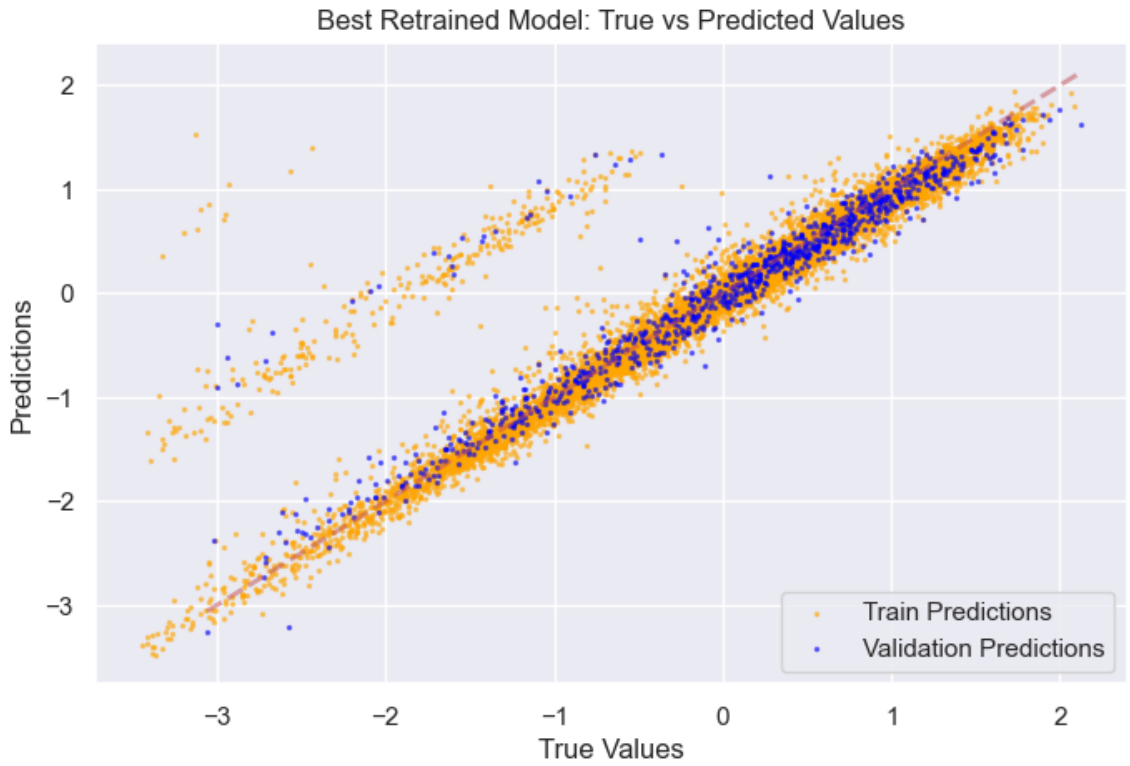

Figure 17: Comparison of true vs. predicted values in the final best model

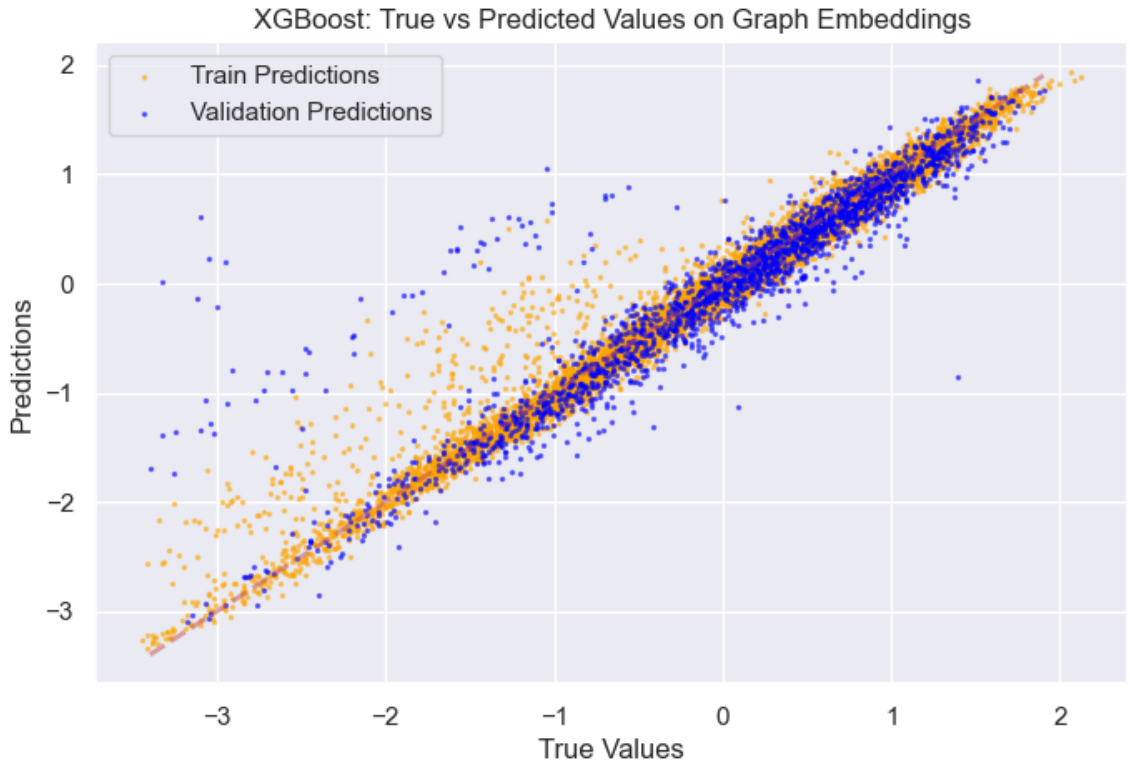

Figure 18: Comparison of true vs. predicted values in the xgboost trained on the model extracted features

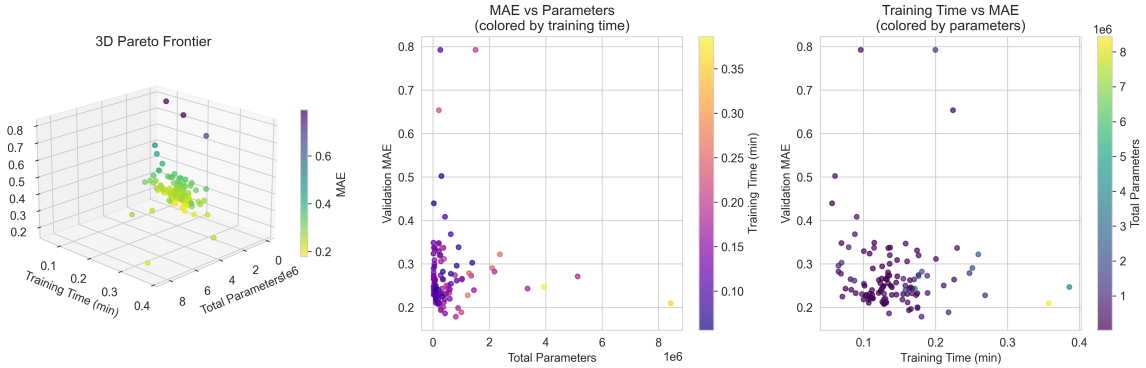

Figure 19: Three-dimensional Pareto frontier analysis showing the trade-offs between validation MAE, model parameters, and training time. The plot reveals optimal configurations that balance accuracy, model complexity, and computational efficiency.

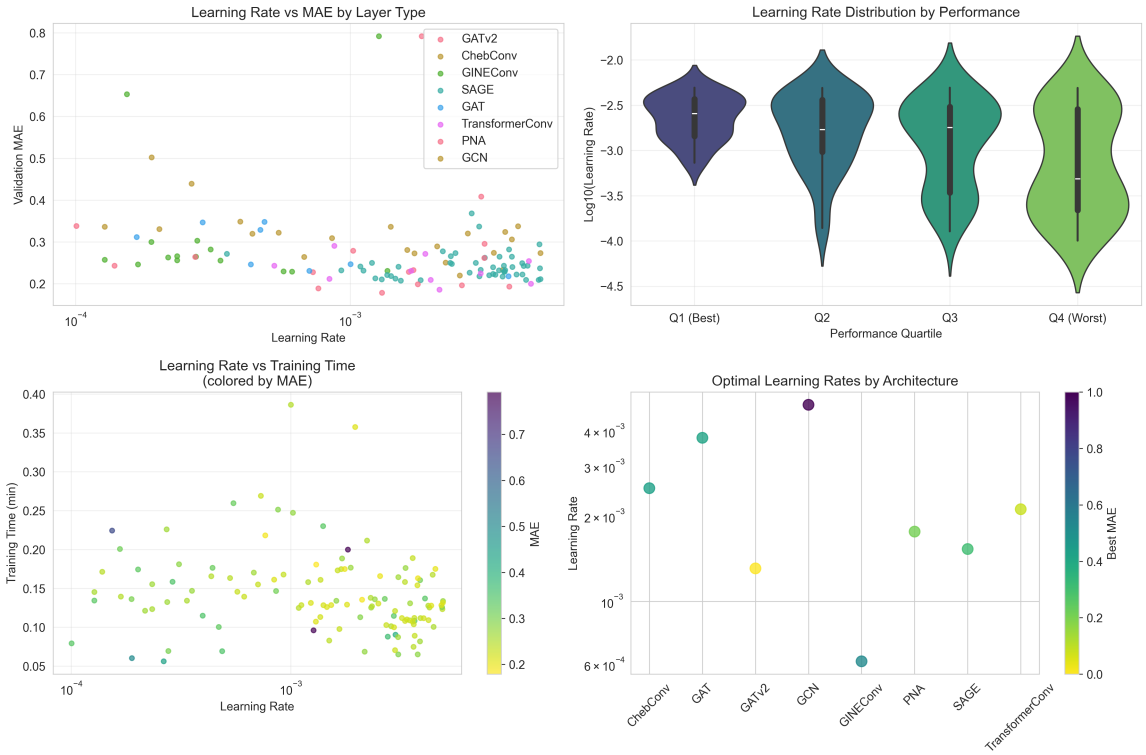

Figure 20: Learning rate sensitivity analysis across different GNN architectures. The four-panel visualization shows (a) learning rate vs MAE by layer type, (b) learning rate distribution by performance quartiles, (c) learning rate vs training time relationship, and (d) optimal learning rate ranges for each architecture type.

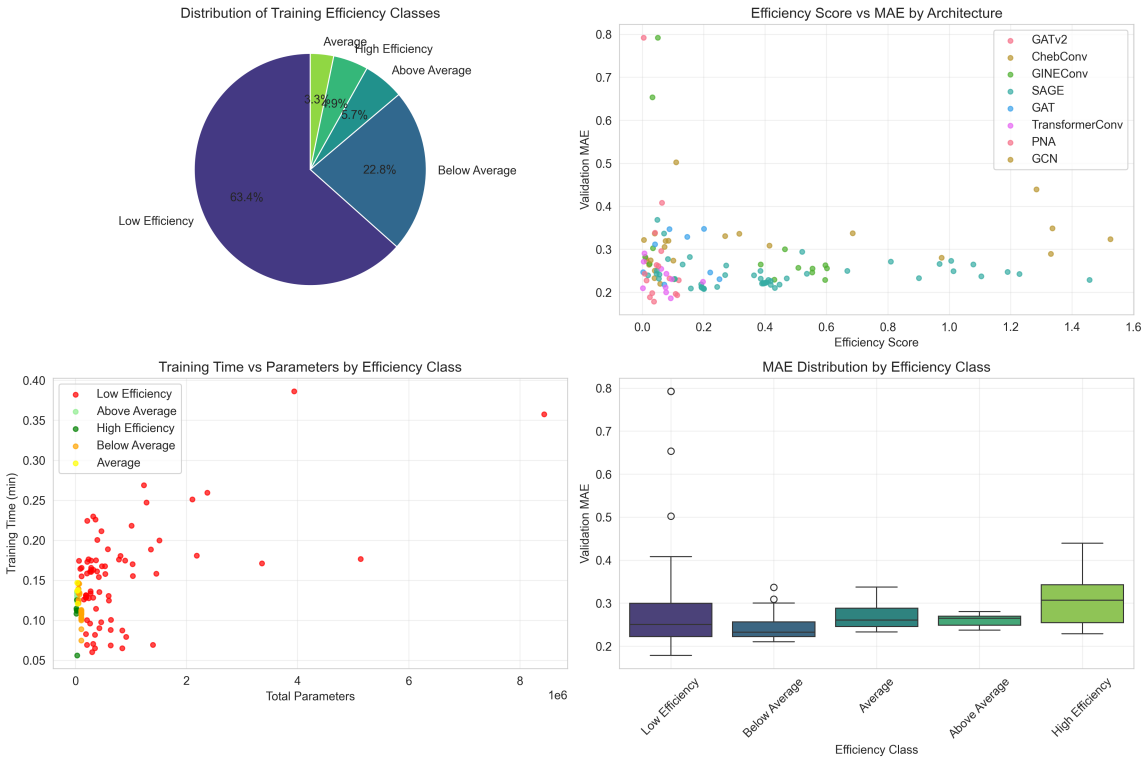

Figure 21: Training efficiency analysis categorizing models into efficiency classes based on computational cost-performance trade-offs. The efficiency is a composite metric calculated as $1/(MAE Training_Time Parameters/1000)$. Higher scores indicate better efficiency: models that achieve lower error with less training time and fewer parameters. This score balances accuracy, computational cost, and model complexity to identify the most resource-efficient configurations. The visualization includes (a) efficiency class distribution, (b) efficiency score vs MAE by architecture, (c) training time vs parameters by efficiency class, and (d) MAE distribution across efficiency classes.

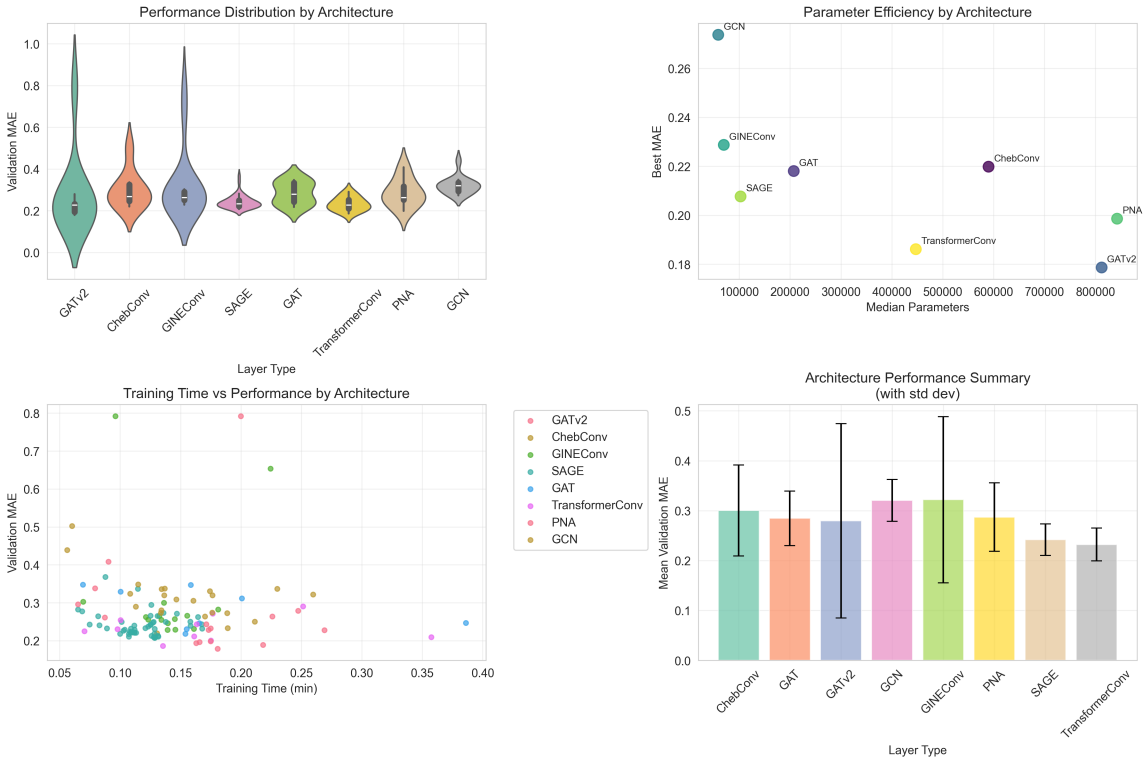

Figure 22: Comprehensive comparison of GNN architecture performance showing (a) MAE distribution by architecture type, (b) parameter efficiency analysis, (c) training time vs performance scatter plot, and (d) architecture performance summary with statistical significance indicators.

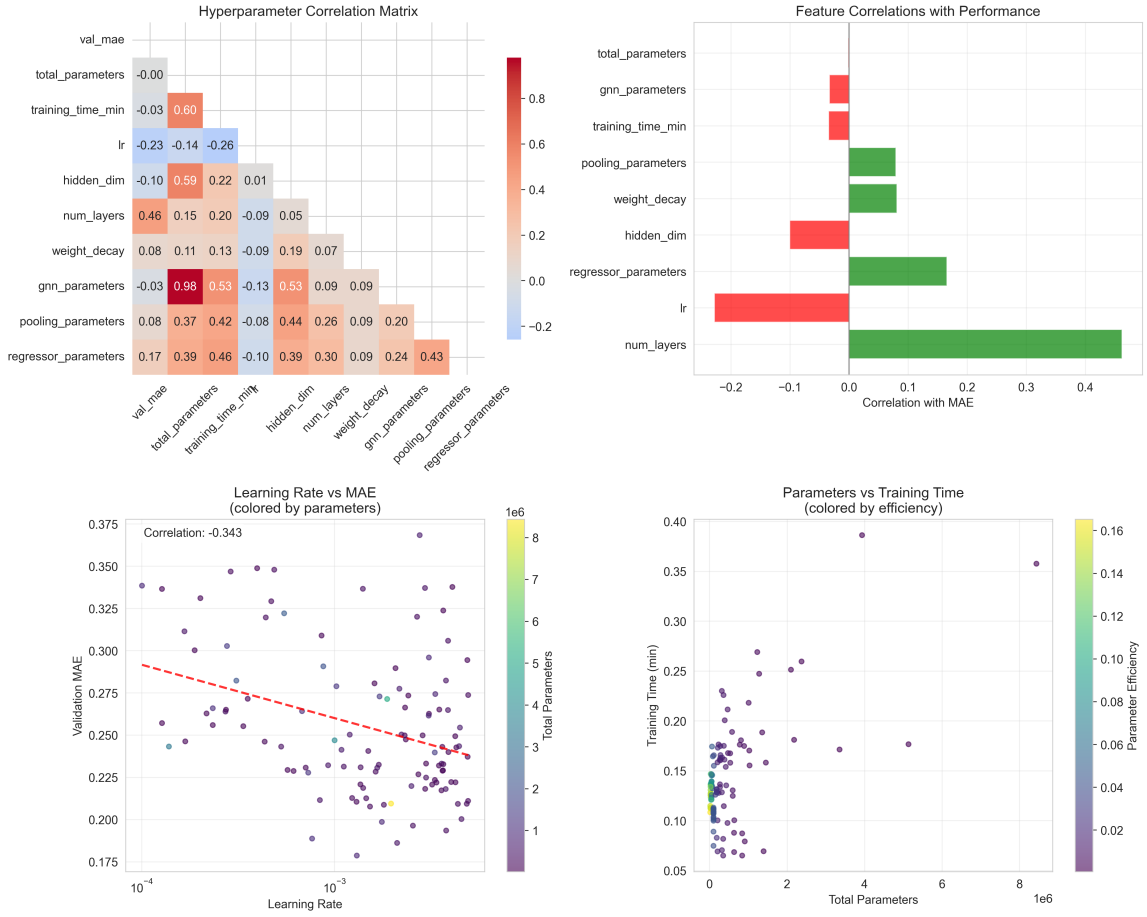

Figure 23: Hyperparameter correlation matrix and performance impact analysis. The visualization reveals (a) correlation heatmap between all hyperparameters, (b) feature correlations with MAE performance, (c) learning rate vs MAE relationship, and (d) parameter count vs training time efficiency mapping.

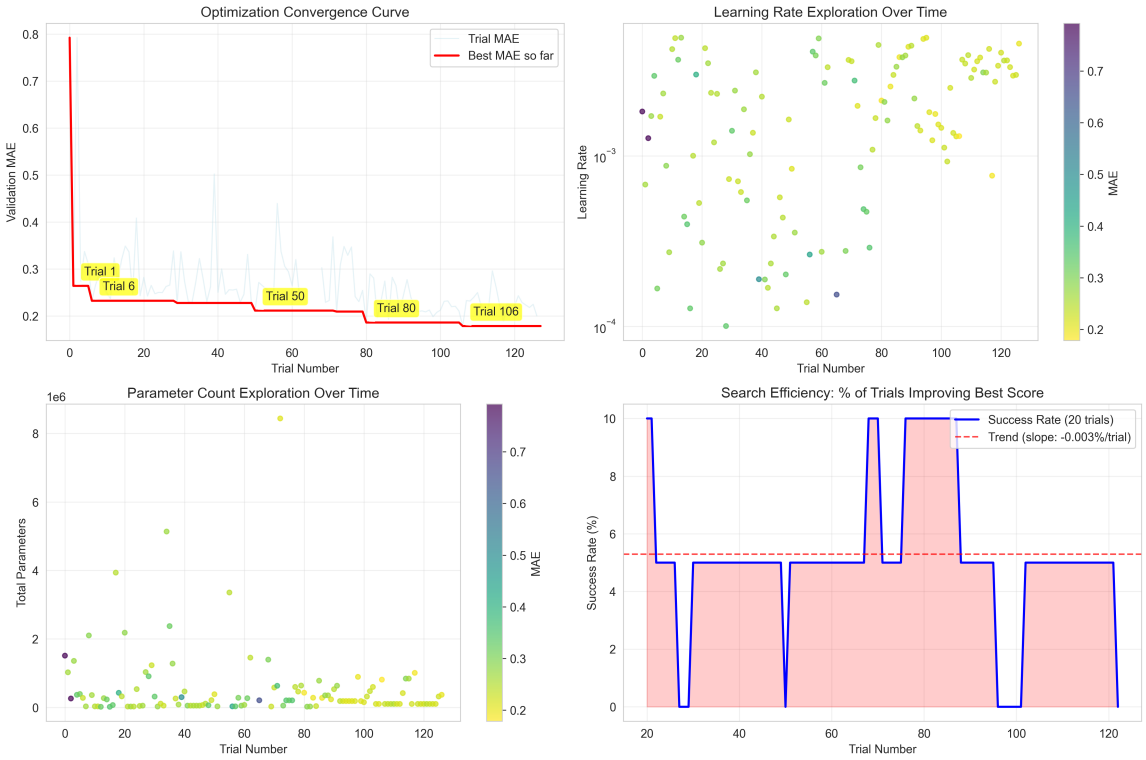

Figure 24: Optuna optimization convergence analysis showing (a) convergence curve with trial-by-trial MAE and cumulative best performance, (b) improvement frequency and plateau detection, (c) quartile performance analysis over time, and (d) convergence efficiency metrics with key milestone annotations.

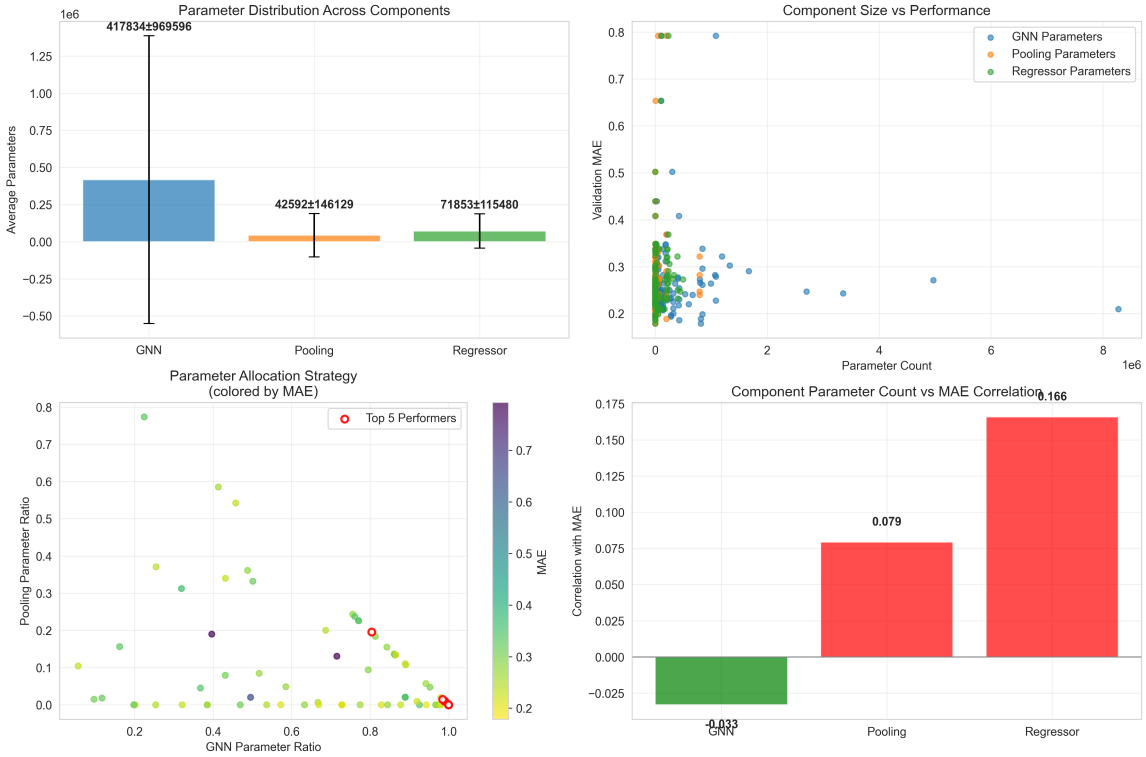

Figure 25: Pipeline component parameter distribution and impact analysis. The four-panel visualization shows (a) parameter distribution across GNN, pooling, and regressor components, (b) component size vs performance relationship, (c) parameter allocation strategy analysis, and (d) correlation analysis between component parameter counts and validation MAE.

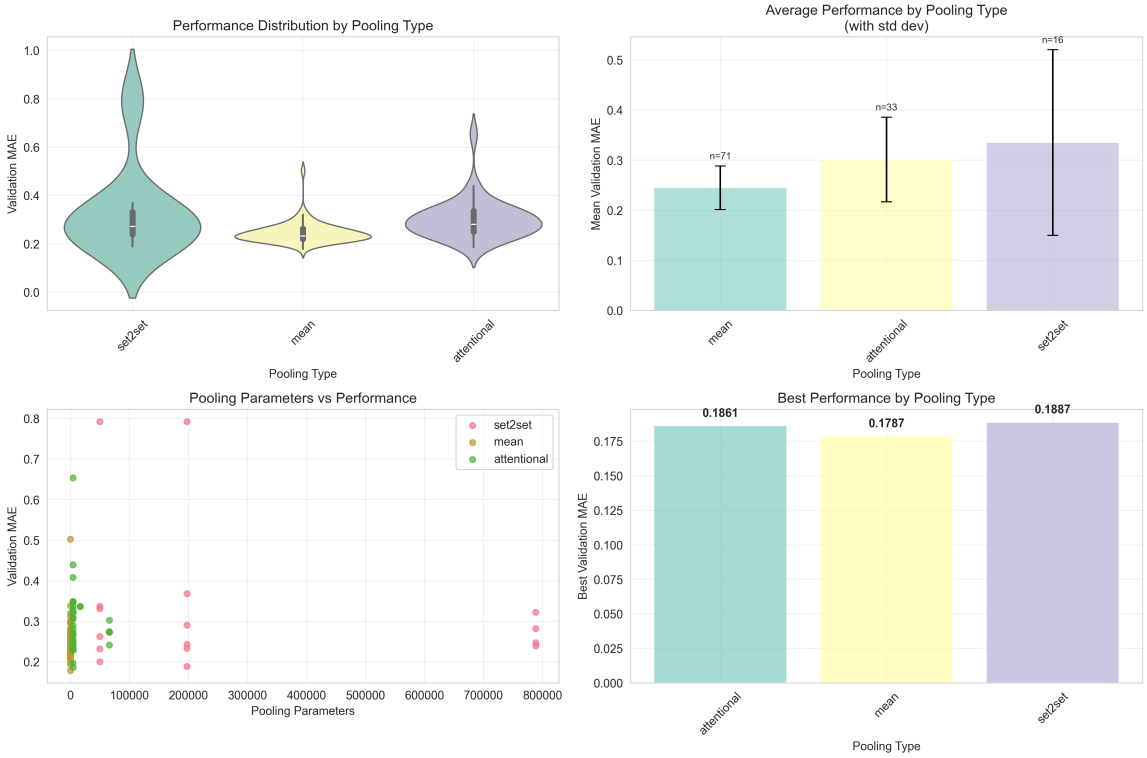

Figure 26: Statistical analysis of pooling strategy performance across all optimization trials. The analysis reveals (a) MAE distribution by pooling type via violin plots, (b) mean performance comparison with confidence intervals, (c) pooling parameter count vs performance relationship, and (d) best MAE achievement by pooling strategy. Statistical significance confirmed via Kruskal-Wallis test ($p < 0.001$), with mean pooling achieving the best performance (MAE: 0.1787).

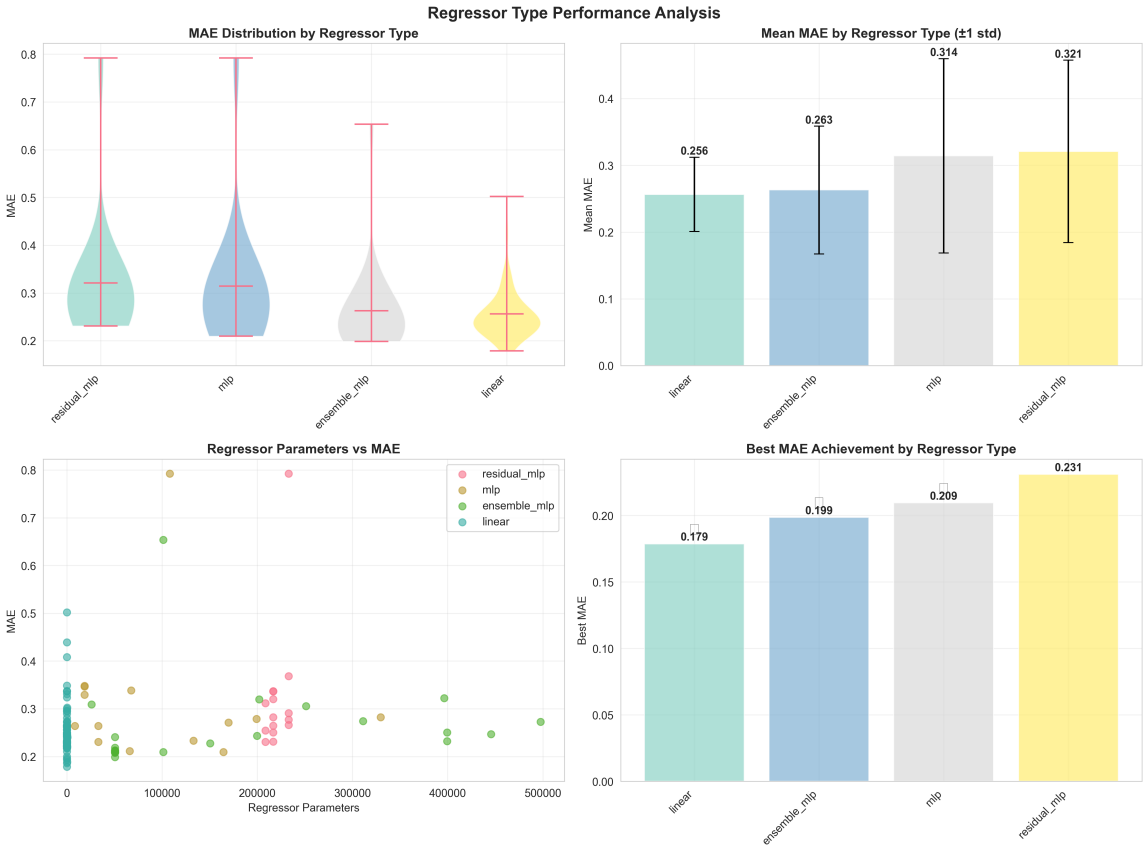

Figure 27: Comprehensive regressor architecture performance evaluation based on 120 trials with complete regressor type information. The analysis includes (a) MAE distribution by regressor type, (b) mean performance comparison with standard deviations, (c) regressor parameter count vs performance scatter plot, and (d) best MAE achievement ranking. Statistical analysis (Kruskal-Wallis test, $p = 0.003$) confirms significant performance differences, with linear regressors achieving optimal results (MAE: 0.1787).